# Nona Esercitazione

Thread e memoria condivisa
Sincronizzazioe tramite semafori

Andrea Reale
andrea.reale@unibo.it

## Agenda

- **Esercizio 1**
  - Sincronizzazione tra thread tramite intrinsic lock (*synchronized*) e primitive *Object.wait()* e *Object.notify()*
- **Esercizio 2**
  - Sincronizzazione avanzata produttore consumatore tramite *semafori*

# Tabella Riassuntiva

| | |
|---|---|
| **synchronized** | • **Keyword di linguaggio**<br>• ***synchronized block***: acquisizione del lock di mutua esclusione *dell'oggetto specificato nello statement*, e rilascio automatico alla fine del blocco<br>• ***synchronized method***: acquisizione del lock di mutua esclusione *dell'istanza su cui si sta invocando il metodo*, e rilascio automatico alla fine del blocco |
| **wait()** | • **Rilascia** il lock intrinseco dell'oggetto su cui wait() è invocato<br>• Inserisce il thread chiamante nel **wait set** dell'oggetto<br>• Quando risvegliato, il thread deve riacquisire il lock |
| **notify()** | • **Risveglia** un thread (quale?) dal **wait set** dell'oggetto su cui è invocato<br>• Il chiamante deve possedere il lock dell'oggetto su cui notify è invocato. Tale lock non viene rilasciato al termine della chiamata. (**signal & continue**) |
| **notifyAll()** | • Come notify(), ma vengono risvegliati **tutti** i thread nel wait set. Quale continua? |

# Esercizio 1

Sincronizzazione tra thread con
**synchronized**, **wait** e **notify**

# Esercizio 1 - Traccia (1/4)

Si realizzi un programma che tramite i thread in Java implemente li seguenti classi.

**Magazzino**
Un magazzino può contenere **due tipi di materiale (A o B)**. La classe deve implementare la seguente interfaccia pubblica:

- **void prenota(int quantA, int quantB)**
  - permette ad un cliente di ordinare **quantA** unità di materiale di tipo A **e quantB** materiale di tipo B. Se non è possibile soddisfare **completamente** la richiesta, il thread chiamante deve essere **bloccato** fino a che non sarà possibile consegnare tutto il materiale richiesto.
- **void produci(int tipoMateriale, int quantita);**
  - permette ad un fornitore di depositare nel magazzino **quantita** unità di materiale. Il tipo del materiale è specificato tramite il parametro **tipoMateriale** **(1 o 2)**.

# Esercizio 1 - Traccia (2/4)

**Cliente**
Istanze di questa classe devono essere eseguibili da propri thread di controllo (oggetti attivi). La classe implemente la seguente interfaccia pubblica:

- **Cliente(Magazzino m, int quantA, int quantB)**
  - Costruttore di classe. Nei parametri specifica i **dettagli dell'ordine** che sarà effettuato dal cliente dal magazzino **m**.
- **void run()**
  - *Entry point* del thread che esegue il cliente. Effettua l'ordine specificato in fase di costruzione **e poi termina.**

# Esercizio 1 - Traccia (3/4)

**Fornitore**
Istanze di Fornitore sono **oggetti attivi** che **periodicamente** consegnano materiale di un certo tipo ad un magazzino
La classe implementa la seguente interfaccia pubblica:

- **Fornitore(Magazzino m, int t, int quant, long periodo)**
  - Istanzia un nuovo Fornitore, specificando i dettagli del suo comportamento. In particolare tale istanza consegnerà materiale di tipo **t (1 o 2)** in quantità **quant** al magazzino **m**. La consegna deve essere ripetuta ogni **periodo** millisecondi.
- **void run()**
  - *Entry point* del thread che esegue il codice del fornitore. Tale thread dovrà periodicamente consegnare al magazzino la quantità specificata di materiale, **fin quando** il metodo stopFornitore()non verrà invocato.
- **void stopFornitore()**
  - I**nvocato da un thread diverso** da quello che esegue il codice del fornitore (*es. Thread-main*), dovrà indicare al thread del fornitore di interrompere le forniture.

# Esercizio 1 - Traccia (4/4)

- Si scriva, infine, un metodo **main** che istanzi un oggetto **Magazzino** e **due** oggetti **Fornitore** (uno per tipo di materiale).
  - Le forniture devono essere avviate tramite appositi meccanismi di creazione di Thread.

Inoltre, il main dovrà creare 10 oggetti **Cliente**, ciascuno dei quali dovrà essere eseguito dal suo thread di controllo.

  - Tra la creazione di un **Cliente** ed il successivo il *Thread-main* dovrà attendere un intervallo di tempo casuale **compreso tra 0 e 500 millisecondi.**
- Dopo aver creato i Clienti, il *Thread-main* dovrà **attendere** che tutti abbiano terminato e ricevuto il proprio ordine.
  - A questo punto il *Thread-main* istruirà i due oggetti **Fornitore** di **terminare la propria attività**.

# Note alla soluzione

- Si relizzino le classi e i comportamenti specificati, complementando le classi descritte con i metodi e variabili di classe non pubblici che si ritengono opportuni.
- Si realizzi la sincronizzazione tra thread (mutua esclusione e attese) tramite i meccanismi di base del linguaggio **synchronized/wait/notify**

- **Problemi principali:**
  - Quanti e quali thread accedono a **risorse** (quali?) in maniera potenzialmente **concorrente**? Regolarne l'accesso in maniera opportuna.
  - Cosa succede se un cliente non trova materiale a sufficienza per soddisfare il suo ordine? Come si fa a **bloccarlo**? *Quali sono le condizioni per cui debba riprendere l'esecuzione?*
  - Come indicare (da un altro thread) ad un Fornitore di terminare in maniera *"pulita"*. **Idee?** (No Thread.stop() -> perché?)

# Altre info utili

**java.util.Random:**
Oggetto per la creazione di distribuzioni uniformi di numeri casuali

**Random(long seed)**
  - Istanzia un generatore di numeri casuali a partire dal seed indicato
- **int nextInt(int n)**
  - Restituisce un numero casuale uniformemente distribuito **nell'intervallo [0, n)**

**Per il resto dei metodi di Random, e per tutta la ricca API delle classi distribuite incluse nella JDK (Java 6 SE):**
- **http://www-lia.deis.unibo.it/Misc/SW/Java/java1.6/docs/api/**
- (oppure) **http://download.oracle.com/javase/6/docs/api/**

# Soluzione - Classe Magazzino

```
public class Magazzino {
   private int q1 = 0
   private int q2 = 0;
   // Lock di mutuea esclusione per
   // l'accesso ai materiali magazzino
   private final Object mutex = new Object();

   // CONTINUA...
```

# Soluzione - Classe Magazzino

```
// CONTINUA MAGAZZINO
  public void compra(int quant1, int quant2) {
     synchronized (mutex) {
        try {
           while (this.q1 < quant1
                 || this.q2 < quant2) {
              this.mutex.wait();
           }
           Thread.sleep(500); // Processo l'ordine
           this.q1 -= quant1;
     this.q2 -= quant2;
        } catch (InterruptedException e) {
           System.err.println(
             Thread.currentThread().getName()
      + " interrotto metre era in attesa di"
              + " una prenotazione. Annullata.");
  } // Fine Try-Catch
     } // Fine sezione critica
  }
      // CONTINUA...
```

# Soluzione - Classe Magazzino

```
// CONTINUA MAGAZZINO
    public void produci(int tipo, int amount) {
        synchronized (mutex) {
            if (tipo == 1) {
        this.quantitaMateriale1 += amount;
    } else if (tipo == 2) {
        this.quantitaMateriale2 += amount;
    }

    // Perché è necessario notifyAll
            // e non basta notify?
    this.mutex.notifyAll();
    }
} // Fine classe Magazzino
```

# Soluzione - Classe Cliente

```
public class Cliente implements Runnable {
    private final int id, quant1, quant2;
    private final Magazzino sorgente;

    public Cliente(int id, int q1, int q2,
                Magazzino sorgente) {
this.id = id;
this.quant1 = quantita1;
this.quant2 = quantita2;
this.sorgente = sorgente;
    }

    @Override
    public void run() {
        // Chiamata bloccante
        this.sorgente.compra(this.q1, this.q2);
    }
}
```

# Soluzione - Classe Fornitore

```
public class Fornitore implements Runnable {
   private final int tipo, quant;
   private final long periodo;
   private final Magazzino dest;

   // Variabile usata per la condizione di terminazione
   // del fornitore. Accesso mutuamente esclusivo!
   private boolean inFornitura;

   public Fornitore( Magazzino m, int t,
        long period, int quant) {
      this.tipo = t;
      this.periodo = period;
      this.quant = quant;
this.inFornitura = true;
this.dest = m;
   }
}
```

# Soluzione - Classe Fornitore

```
  public synchronized boolean isInFornitura() {
     return this.inFornitura;
  }
  public synchronized void stopFornitura() {
     this.inFornitura = false;
  }
  public void run() {
     while (this.isInFornitura()) {
  this.dest.produci(tipo, quant);
  try {
            Thread.sleep(periodo);
  } catch (InterruptedException e) {
            System.err.println("Interruzione!");
     break;
  }
}
  }
} // Fine Fornitore
```

# Una variante

- E' strettamente necessario che **Fornitori** che producono **materiali diversi** per il magazzino accedano ad esso in maniera strettamente esclusiva?
- **Riconsiderare** i requisiti di sincronizzazione "incrociati" dei diversi attori (oggetti **Fornitore** e **Cliente**) e - se possibile - rielaborare la soluzione permettendo una *sequenzializzazione delle azioni meno stretta*.

# Esercizio 2

Sincronizzazione tra processi
tramite Semaforo in Java

# Realizzazione della primitiva Semaforo in Java

- Creazione di una struttura dati Semaforo che realizza il comportamento del semaforo
- **Semafori** visti **abbondantemente** a lezione
- Da Java 1.5 SE semaforo introdotto nella JDK
  - **java.util.concurrent.Semaphore** (vedi Java API)
- Noi utilizzeremo l'implementazione "manuale" del semaforo vista a lezione
  - Realizzata tramite i costrutti **synchronized**, **wait** e **notify**

# Classe Semaforo

```
public class Semaforo {
   private int value;

   public Semaforo(int initial) {
      if (initial < 0)
         throw new IllegalArgumentException();
      this.value = initial;
   }

   public synchronized void p() throws
                  InterruptedException {
      while (this.value == 0) {
         this.wait();
      }
      value--;
   }

   public synchronized void v() {
      if ( value == 0 ) this.notify();
      value++;
   }
}
```

# Esercizio 2 - Traccia (1/2)

Si realizzi un programma Java che simuli la gestione di una cucina di un ristorante.

In particolar modo:

- Due thread **AiutoCuoco** si incaricano di portare sul tavolo di preparazione (**risorsa condivisa**) gli ingredienti necessari (di due tipi diversi)

- Un thread **Chef** sovrintende alla preparazione del piatto,
  - **Attende** che siano disponibili le quantità necessarie di ingredienti
  - **Quando disponibili**, le preleva dal tavolo e prepara il piatto

# Esercizio 2 - Traccia (2/2)

- **Tavolo** ha **capacità limitata** (*diversa*) per ingrediente1 e ingrediente2 (risp. **M1** e **M2**)

- Ciascun **AiutoCuoco** porta periodicamente una unità di ingrediente

- Per preparare il piatto, **Chef** ha bisogno **di quantità prestabilite** per i 2 ingredienti
  - Q1 per ingrediente1
  - Q2 per ingrediente2

# Esercizio 2 - Sincronizzazione

- **Tavolo** è una risorsa condivisa con *due buffer*, uno per ciascun ingrediente
  - **gestione della capacità**: *non è possibile altre unità di ingredienteX se il tavolo ha già saturato la capacità per X*
- Necessità di **mutua esclusione** tra thread che accedono al tavolo
- Thread **Chef** deve attendere che siano disponibili gli ingredienti
  - Soltanto allora comincia la preparazione del piatto (**ordinamento** tra thread)
  - Si assuma che **il tavolo sia inizialmente vuoto**

# Esercizio 2 - Alcune Note

- **Si utilizzino i semafori** (classe **Semaforo**) per risolvere *tutti* i problemi di sincronizzazione tra thread
  - **Mutua esclusione** nell'accesso ad una risorsa condivisa
  - **Gestione della capacità** dei buffer ingredienti
  - **Ordinamento** di operazioni di thread diversi (lo Chef attende che ci siano abbastanza ingredienti prima di iniziare a cucinare)

- Per semplicità, si assuma che le operazioni degli **AiutoCuoco** e dello **Chef** si ripetano un numero *indefinito* di volte nel tempo.
  - I thread continuano ad operare fin quando la JVM che li esegue non viene interrotta dall'utente "a mano"